SOFTWARE AND MIND

Andrei Sorin

EXTRACT

Chapter 7: *Software Engineering*
Section *The Relational Database Model*
Subsections *The Basic File Operations, The Lost Integration*

These subsections examine the traditional operations involving indexed data files, their integration with programming languages, and their benefits relative to relational databases.

The entire book, each chapter separately, and also selected sections, can be viewed and downloaded at the book's website.

www.softwareandmind.com

SOFTWARE AND MIND

The Mechanistic Myth and Its Consequences

Andrei Sorin

ANDSOR BOOKS

Published by Andsor Books, Toronto, Canada (January 2013)
www.andsorbooks.com

For disclaimers see pp. vii, xv–xvi.

Designed and typeset by the author with text management software developed by the author and with Adobe FrameMaker 6.0. Printed and bound in the United States of America.

Acknowledgements

Excerpts from the works of Karl Popper: reprinted by permission of the University of Klagenfurt/Karl Popper Library.

Excerpts from *The Origins of Totalitarian Democracy* by J. L. Talmon: published by Secker & Warburg, reprinted by permission of The Random House Group Ltd.

Excerpts from *Nineteen Eighty-Four* by George Orwell: Copyright ©1949 George Orwell, reprinted by permission of Bill Hamilton as the Literary Executor of the Estate of the Late Sonia Brownell Orwell and Secker & Warburg Ltd.; Copyright ©1949 Harcourt, Inc. and renewed 1977 by Sonia Brownell Orwell, reprinted by permission of Houghton Mifflin Harcourt Publishing Company.

Excerpts from *The Collected Essays, Journalism and Letters of George Orwell*: Copyright ©1968 Sonia Brownell Orwell, reprinted by permission of Bill Hamilton as the Literary Executor of the Estate of the Late Sonia Brownell Orwell and Secker & Warburg Ltd.; Copyright ©1968 Sonia Brownell Orwell and renewed 1996 by Mark Hamilton, reprinted by permission of Houghton Mifflin Harcourt Publishing Company.

Excerpts from *Doublespeak* by William Lutz: Copyright ©1989 William Lutz, reprinted by permission of the author in care of the Jean V. Naggar Literary Agency.

Excerpts from *Four Essays on Liberty* by Isaiah Berlin: Copyright ©1969 Isaiah Berlin, reprinted by permission of Curtis Brown Group Ltd., London, on behalf of the Estate of Isaiah Berlin.

Library and Archives Canada Cataloguing in Publication

Sorin, Andrei
Software and mind : the mechanistic myth and its consequences / Andrei Sorin.

Includes index.
ISBN 978-0-9869389-0-0

1. Computers and civilization. 2. Computer software – Social aspects.
3. Computer software – Philosophy. I. Title.

QA76.9.C66S67 2013 303.48'34 C2012-906666-4

Printed on acid-free paper.

Don't you see that the whole aim of Newspeak is to narrow the range of thought?… Has it ever occurred to you … that by the year 2050, at the very latest, not a single human being will be alive who could understand such a conversation as we are having now?

George Orwell, *Nineteen Eighty-Four*

Disclaimer

This book attacks the mechanistic myth, not persons. Myths, however, manifest themselves through the acts of persons, so it is impossible to discuss the mechanistic myth without also referring to the persons affected by it. Thus, all references to individuals, groups of individuals, corporations, institutions, or other organizations are intended solely as examples of mechanistic beliefs, ideas, claims, or practices. To repeat, they do not constitute an attack on those individuals or organizations, but on the mechanistic myth.

Except where supported with citations, the discussions in this book reflect the author's personal views, and the author does not claim or suggest that anyone else holds these views.

The arguments advanced in this book are founded, ultimately, on the principles of demarcation between science and pseudoscience developed by philosopher Karl Popper (as explained in "Popper's Principles of Demarcation" in chapter 3). In particular, the author maintains that theories which attempt to explain non-mechanistic phenomena mechanistically are pseudoscientific. Consequently, terms like "ignorance," "incompetence," "dishonesty," "fraud," "corruption," "charlatanism," and "irresponsibility," in reference to individuals, groups of individuals, corporations, institutions, or other organizations, are used in a precise, technical sense; namely, to indicate beliefs, ideas, claims, or practices that are mechanistic though applied to non-mechanistic phenomena, and hence pseudoscientific according to Popper's principles of demarcation. In other words, these derogatory terms are used solely in order to contrast our world to a hypothetical, ideal world, where the mechanistic myth and the pseudoscientific notions it engenders would not exist. The meaning of these terms, therefore, must not be confused with their informal meaning in general discourse, nor with their formal meaning in various moral, professional, or legal definitions. Moreover, the use of these terms expresses strictly the personal opinion of the author – an opinion based, as already stated, on the principles of demarcation.

This book aims to expose the corruptive effect of the mechanistic myth. This myth, especially as manifested through our software-related pursuits, is the greatest danger we are facing today. Thus, no criticism can be too strong. However, since we are all affected by it, a criticism of the myth may cast a negative light on many individuals and organizations who are practising it unwittingly. To them, the author wishes to apologize in advance.

Contents

Preface

The book's subtitle, *The Mechanistic Myth and Its Consequences*, captures its essence. This phrase is deliberately ambiguous: if read in conjunction with the title, it can be interpreted in two ways. In one interpretation, the mechanistic myth is the universal mechanistic belief of the last three centuries, and the consequences are today's software fallacies. In the second interpretation, the mechanistic myth is specifically today's mechanistic *software* myth, and the consequences are the fallacies *it* engenders. Thus, the first interpretation says that the past delusions have caused the current software delusions; and the second one says that the current software delusions are causing further delusions. Taken together, the two interpretations say that the mechanistic myth, with its current manifestation in the software myth, is fostering a process of continuous intellectual degradation – despite the great advances it made possible. This process started three centuries ago, is increasingly corrupting us, and may well destroy us in the future. The book discusses all stages of this degradation.

The book's epigraph, about Newspeak, will become clear when we discuss the similarity of language and software (see, for example, pp. 411–413).

Throughout the book, the software-related arguments are also supported with ideas from other disciplines – from philosophy, in particular. These discussions are important, because they show that our software-related problems

are similar, ultimately, to problems that have been studied for a long time in other domains. And the fact that the software theorists are ignoring this accumulated knowledge demonstrates their incompetence. Often, the connection between the traditional issues and the software issues is immediately apparent; but sometimes its full extent can be appreciated only in the following sections or chapters. If tempted to skip these discussions, remember that our software delusions can be recognized only when investigating the software practices from this broader perspective.

Chapter 7, on software engineering, is not just for programmers. Many parts (the first three sections, and some of the subsections in each theory) discuss the software fallacies in general, and should be read by everyone. But even the more detailed discussions require no previous programming knowledge. The whole chapter, in fact, is not so much about programming as about the delusions that pervade our programming practices. So this chapter can be seen as a special introduction to software and programming; namely, comparing their true nature with the pseudoscientific notions promoted by the software elite. This study can help both programmers and laymen to understand why the incompetence that characterizes this profession is an inevitable consequence of the mechanistic software ideology.

There is some repetitiveness in the book, deliberately introduced in order to make the individual chapters, and even the individual sections, reasonably independent. Thus, while the book is intended to be read from the beginning, you can select almost any portion and still follow the discussion. An additional benefit of the repetitions is that they help to explain the more complex issues, by presenting the same ideas from different perspectives or in different contexts.

The book is divided into chapters, the chapters into sections, and some sections into subsections. These parts have titles, so I will refer to them here as *titled* parts. Since not all sections have subsections, the lowest-level titled part in a given place may be either a section or a subsection. This part is, usually, further divided into *numbered* parts. The table of contents shows the titled parts. The running heads show the current titled parts: on the right page the lowest-level part, on the left page the higher-level one (or the same as the right page if there is no higher level). Since there are more than two hundred numbered parts, it was impractical to include them in the table of contents. Also, contriving a short title for each one would have been more misleading than informative. Instead, the first sentence or two in a numbered part serve also as a hint of its subject, and hence as title.

Figures are numbered within chapters, but footnotes are numbered within the lowest-level titled parts. The reference in a footnote is shown in full only the first time it is mentioned within such a part. If mentioned more than once,

in the subsequent footnotes it is usually abbreviated. For these abbreviations, then, the full reference can be found by searching the previous footnotes no further back than the beginning of the current titled part.

The statement "italics added" in a footnote indicates that the emphasis is only in the quotation. Nothing is stated in the footnote when the italics are present in the original text.

In an Internet reference, only the site's main page is shown, even when the quoted text is from a secondary page. When undated, the quotations reflect the content of these pages in 2010 or later.

When referring to certain individuals (software theorists, for instance), the term "expert" is often used mockingly. This term, though, is also used in its normal sense, to denote the possession of true expertise. The context makes it clear which sense is meant.

The term "elite" is used to describe a body of companies, organizations, and individuals (for example, the software elite); and the plural, "elites," is used when referring to several entities, or groups of entities, within such a body. Thus, although both forms refer to the same entities, the singular is employed when it is important to stress the existence of the whole body, and the plural when it is the existence of the individual entities that must be stressed. The plural is also employed, occasionally, in its normal sense – a group of several different bodies. Again, the meaning is clear from the context.

The issues discussed in this book concern all humanity. Thus, terms like "we" and "our society" (used when discussing such topics as programming incompetence, corruption of the elites, and drift toward totalitarianism) do not refer to a particular nation, but to the whole world.

Some discussions in this book may be interpreted as professional advice on programming and software use. While the ideas advanced in these discussions derive from many years of practice and from extensive research, and represent in the author's view the best way to program and use computers, readers must remember that they assume all responsibility if deciding to follow these ideas. In particular, to apply these ideas they may need the kind of knowledge that, in our mechanistic culture, few programmers and software users possess. Therefore, the author and the publisher disclaim any liability for risks or losses, personal, financial, or other, incurred directly or indirectly in connection with, or as a consequence of, applying the ideas discussed in this book.

The pronouns "he," "his," "him," and "himself," when referring to a gender-neutral word, are used in this book in their universal, gender-neutral sense. (Example: "If an individual restricts himself to mechanistic knowledge, his performance cannot advance past the level of a novice.") This usage, then, aims solely to simplify the language. Since their antecedent is gender-neutral ("everyone," "person," "programmer," "scientist," "manager," etc.), the neutral

sense of the pronouns is established grammatically, and there is no need for awkward phrases like "he or she." Such phrases are used in this book only when the neutrality or the universality needs to be emphasized.

It is impossible, in a book discussing many new and perhaps difficult concepts, to anticipate all the problems that readers may face when studying these concepts. So the issues that require further discussion will be addressed online, at *www.softwareandmind.com*. In addition, I plan to publish there material that could not be included in the book, as well as new ideas that may emerge in the future. Finally, in order to complement the arguments about traditional programming found in the book, I plan to publish, in source form, some of the software applications I developed over the years. The website, then, must be seen as an extension to the book: any idea, claim, or explanation that must be clarified or enhanced will be discussed there.

The Basic File Operations

1

To appreciate the inanity of the relational model, we must start by examining the basic file operations; that is, those operations which the relational systems are attempting to supplant. What I want to show is that these operations are *both necessary and sufficient* for implementing database management requirements, particularly in business applications. Thus, once we recognize the importance of the basic file operations, we will be in a better position to understand why the relational systems are fraudulent. For, as we will see, the only way to make them useful was by enhancing them with precisely those capabilities provided by the basic file operations; in other words, by restoring the very features that the database experts had claimed to be unnecessary.

Also, it is important to remember that the basic file operations have been available to programmers from the start, ever since mass storage devices with random access became popular. (They are sometimes called ISAM, Indexed Sequential Access Method.) For example, they have been available through COBOL (a language specifically designed for business applications) since around 1970. So these operations have always been well known: COBOL was always a public language, was implemented on all major computers, and was adopted by most companies. Thus, in addition to being an introduction to the basic file operations, this discussion serves to support my claim that the only motivation for database systems in general, and for the relational systems in particular, was to find a substitute for the knowledge required of programmers to use these operations correctly.

❖

Before examining the basic file operations, we must take a moment to clarify this term and the related terms "file operations" and "database operations." The basic file operations are a basic set of file management functions. They formed in the past an integral part of every major operating system, and were accessible through programming languages. These operations deal with *indexed data files* – the most versatile form of data storage; and, in conjunction with the features provided by the languages themselves, they allow us to use and to relate these files in any way we like.

"File operations" is a more general term. It refers to the basic file operations, but also to the various ways in which we combine them, using the flow-control constructs of a programming language, in order to implement file management requirements. "Database operations" is an even more general

term. It refers to the file operations, but in the context of the whole application, so it usually means *combinations* of file operations; in particular, combinations involving several files. The terms "traditional file operations" and "low-level file operations" refer to any one of the operations defined above.

The term "database" refers to a set of related files; typically, the files used by a particular application. Hence, the term "database system" ought to mean any software system that helps us to manage a database.[1] Through their propaganda, though, the software elites have created in our minds a strong association between terms like "database," "database system," and "database management system" (or DBMS) and *high-level* database operations. And as a result, most people believe that the only way to manage a database is through high-level operations; that the current database systems provide indispensable features; and that it is impossible to implement a serious application without depending on such a system.

But we must not allow the software charlatans to control our language and our minds. Since we can implement any database functions through the basic file operations and a programming language, systems that provide high-level operations are not at all essential for database management. So we can continue to use the terms "database" and "database operations" even while rejecting the notion of a system that restricts us to high-level operations.

Strictly speaking, since the basic file operations permit us to manage a database, they too form a database system. But it would be confusing to use this term for the basic operations, now that it is associated with the high-level operations. Thus, I call the systems that provide basic file operations "*file* management systems," or "*file* systems" for short. This term is quite appropriate, in fact, seeing that these systems are limited to operations involving single files; it is *we* who implement the actual database management, by combining the operations provided by the file system with those provided by a programming language.

So I use the term "database," and terms like "database operations" and "database management," to refer to *any* set of related files – regardless of whether the files and relations are managed through the high-level operations of a *database* system, or through the basic operations of a *file* system.

The term "database structures" refers to the various hierarchical structures created by the files that make up the database: related files can be seen as the levels of a structure, and their records as the elements that make up these levels (see p. 702). In most applications, the totality of database structures is a complex structure.

[1] The term "database system" is used by everyone as an abbreviation of "database management system." It is somewhat misleading, though, since it sounds as if it refers to the database itself.

2

Two types of files make up the database structures of an application: *data* files and *index* files. The data files contain the actual data, organized as *records*; the index files (or indexes, for short) contain the pointers that permit us to access these records.

The record is the unit that the application typically reads from the file, or writes to the file. But within each record the data is broken down into *fields*, and it is the values present in the individual fields that we normally use in the application. For example, if each record in the file has 100 bytes, the first field may take the first 6 bytes, the second one the next 24 bytes, and so on. This is how the fields reside on disk, and in memory when the record is read from disk, but in most cases their relative order within the record is immaterial. For, in the application we assign names to these fields, and we refer to them simply by their names. Thus, once a record is read into memory, we treat database fields, for all practical purposes, as we do memory variables.

The records and fields of a data file reflect the structure and type of the information stored in the file. In an employee file, for example, there is a record for each employee, and each record contains such fields as employee number, name, salary, and year-to-date earnings and deductions; in a sales history file there is a record for each line in a sales order, with such fields as the customer and order numbers, date, price, and quantity sold. While in simple cases the required fields are self-evident, generally it takes some experience to design the most effective database for a given set of requirements. We must decide what information should be processed by the application, how to represent this information, how to distribute it among files, how to index the files, and how to relate them. Needless to say, it is impossible to predict all future requirements, so we must be prepared to alter the application's database structure later: we may need to add or delete fields, move fields from one file to another, and create new files or indexes.

We don't normally access data records directly, but through an index. Indexes, thus, are service files, means to access the data files. Indexes fulfil two essential functions: they allow us to identify a specific record, and to scan a series of records in a specific sequence. It is through *keys* that indexes perform these tasks. The key is one of the fields that make up the record, or a set of several fields. Clearly, if the combination of values present in these fields is different for each record in the file, each record can be uniquely identified. In addition, key uniqueness allows us to scan the records in a particular sequence – the sequence that reflects the current key values – regardless of their actual,

physical sequence on disk. When the key is one field, the value present in the field is the value of the key. When the key consists of several fields, the value of the key is the combination of the field values, in the order in which they make up the key. The records are scanned, in effect, in a sorted sequence. For example, if the key is defined as the set of three fields, *A*, *B*, and *C*, the sorting sequence can be expressed as either "by *A* by *B* by *C*" or "by *C* within *B* within *A*."

Note that if we permit *duplicate* keys – if, that is, some combinations of values in the key fields are not unique – we will be unable to identify the individual records within a set of duplicates. Such an index is still useful, however, if all we need is to *scan* those records. The scanning sequence within a set of duplicate records is usually the order in which they were added to the file. Thus, for scanning too, if we want better control we must ensure key uniqueness.

An especially useful feature is the capability to create several indexes for the same data file. This permits us to access the same records in different ways – scan the file in one sequence or another, or read a record through one key or another. For example, we may scan a sales history file either by order number or by product number; or, we may search for a particular sales record through a key consisting of the customer number and order number, or through a key consisting of the product number and order date.

Another useful indexing feature is the option of *descending* keys. The normal scanning sequence is *ascending*, from low to high key values; but some file systems also allow indexes that scan records from high to low key values. Any one field, or all the fields in the key, can then be either ascending or descending. Simply by scanning the data file through such an index we can list, for instance, orders in ascending sequence by customer number, but within each customer those orders with a higher amount first; or we can list the sales history by ascending product number, but within each product by descending date (so those sold most recently come first), and within each date by ascending customer number. A related indexing feature, useful in its own right but also as an alternative to descending keys, is the capability to scan records backward.

In addition to indexed data files, most file management systems support two other types of files, *relative* and *sequential*. These files provide simpler record access, and are useful for data that does not require an elaborate indexing scheme. In relative data files, we access a record by specifying its relative position in the file (first, second, third, etc.). These files are useful, therefore, in situations where the individual records cannot, or need not, be identified by the values present in their fields (to store the entries of a large table, for instance). Sequential data files are organized as a series of consecutive

records, which can only be accessed sequentially, starting from the beginning. These files are useful in situations where we don't need to access individual records directly, and where we normally read the whole file anyway (to store data that has no specific structure, for instance). Text data, too, is usually stored in sequential files. I will not discuss further the relative and sequential files. It is the indexed data files that interest us, because it is only *their* operations that the relational database systems are attempting to replace with high-level operations.

❖

File systems provide at least two types of fields, *alphanumeric* (or *alpha*, for short) and *numeric*. And, since these types are the same as the memory variables supported by most high-level languages (COBOL, in particular), database fields and memory variables can be used together, and in the same manner, in the application. In alphanumeric fields, data is stored as character symbols, so these fields are useful for names, addresses, descriptions, notes, identifiers, and the like. When these fields are part of an indexing key, the scanning sequence is alphabetical. In numeric fields, the data is stored as numeric values, so these fields can be used directly in calculations. Numeric fields are useful for any data that can be expressed as a numeric value: quantities, dollar amounts, codes, and the like. When part of an indexing key, the scanning sequence is determined by the numeric value.

Some file systems provide additional field types. *Date* fields, for instance, are useful for storing dates. In the absence of date fields, we must store dates in numeric fields, as six- or eight-digit values representing the combination of the month, day, and year; alternatively, we can store dates as values representing the number of days elapsed since some arbitrary, distant date in the past. (The latter method is preferable, as it simplifies date calculations, comparisons, and indexing.) Another field type is the *binary* field, used to store such data as text, graphics, and sound; that is, data which can be in any format whatsoever (hence "binary," or raw), and which may require many thousands of bytes. (Because of its large size, this data is stored in separate files, and only pointers to it are kept in the field itself.)

3

Now that we have examined the structure of indexed data files, let us review the basic file operations. Six operations, combined with the iterative and conditional constructs of high-level languages, are all we need in order to use

indexed data files. I will first describe these operations, and then show how they are combined with language features to implement various requirements. The names I use for the basic operations are taken from COBOL. (There may be some small variations in the way these operations are implemented in a particular file system, or in a particular version of COBOL; for example, in the way multiple indexes or duplicate keys are supported.)

The following terms are used in the description of the file operations: The *current index* is the index file specified in the operation. *File* is a data file; although the file actually specified in the operation is an index file, the record read or written belongs to the data file (we always access a data file through one of its indexes). *Record area* is a storage area – the portion of memory where the fields that make up the record are specified; each file has its own record area, and this area is accessed by both the file system and the application (the application treats the fields as ordinary memory variables). *Key* is the field or set of fields, within the record area, that was defined as the key of a particular index; the *current key* is the key that was defined for the current index. The record *pointer* is an indicator maintained by the file system to identify the next record in the scanning sequence established by a particular index; each index has its own pointer, and the *current pointer* is the pointer corresponding to the current index.

WRITE: A new record is added to the file. Typically, the data in this record consists of the values previously placed by the application into the fields that make up the file's record area. The values present in the fields that make up the current key will become the new record's key in the current index. If the file has additional indexes, the values in their respective key fields will become the keys in those indexes. All indexes are updated together: following this operation, the new record can be accessed either through the current index or through another index. If one of the file's indexes does not permit duplicate keys and the new record would cause such a condition, the operation is aborted and the system returns an error code (so that the application can take appropriate action).

REWRITE: The data in the record area replaces the data in the record currently in the file. Typically, the application read previously the record into the record area through the current index, and modified some of the fields. The record is identified by the current key, so the fields that make up this key should not be modified. If there are other indexes, the fields that make up their keys may be modified, and REWRITE will update those indexes to reflect the change. REWRITE, however, can also be used without first reading the existing record: the application must place some values in all the fields, and REWRITE functions then like WRITE, except that it replaces an existing record. In either case, if no record is found with the current key, or if one of the file's indexes

does not permit duplicate keys and the modified record would cause such a condition, the operation is aborted and the system returns an error code.

DELETE: The record identified by the current key is removed from the file. Only the values present in the current key fields are important for the operation; the rest of the record area is ignored. The application, therefore, can delete a record either by reading it first into the record area (through any one of its indexes) or just by placing the appropriate values into the current key fields. If no record is found with the current key, the system returns an error code.

READ: The record identified by the current key is read into the record area. The current index can be any one of the file's indexes, and only the values present in the current key fields are important for the operation. Following this operation, the fields in the record area contain the values present in that record in the file. If no record is found with the current key, the system returns an error code.

START: The current pointer is positioned at the record identified by the current key. The current index can be any one of the file's indexes, and only the values present in the current key fields are important for the operation. The specification for the operation includes a relation like *equal*, *greater*, or *greater or equal*, so the application need not indicate a valid key; the record identified is simply the first one, in the scanning sequence of the current index, whose key satisfies the condition specified (for example, the first one whose key is *greater* than the values present in the current key fields). If no record in the file satisfies that condition, the system returns an error code.

READ NEXT: The record identified by the current pointer is read into the record area. This operation, in conjunction with START, makes the file scanning feature available to the application. The application must first perform a START for the current index, in order to set the current pointer at the first record in the series of records to be scanned. (To indicate the first record in the file, null values are typically placed in the key fields, and the condition *greater* is specified.) READ NEXT will then read that record and advance the pointer to the next record in the scanning sequence of the current index. The subsequent READ NEXT will read the record indicated by the pointer's new position and advance the pointer to the next record, and so on. Through this process, then, the application can read a series of consecutive records without having to know their keys.[2] Typically, READ NEXT is part of a loop, and the application knows when the last record in the series is reached by checking a certain condition (for example, whether the key exceeds a particular value). If the pointer was already positioned past the last record in the file (the *end-of-file* condition), the

[2] Since no search is involved, it is not only simpler but also faster to read a record in this fashion, than by specifying its key. Thus, even when the keys are known, it is more efficient to read consecutive records with READ NEXT than with READ.

system returns an error code. (Simply checking for this code after each READ NEXT is how applications typically handle the situation where the last record in the series is also the last one in the file.)

❖

These six operations form the minimal practical set of file operations: the set of operations that are both necessary and sufficient for using indexed data files in serious applications.[3] I will demonstrate now, with a few examples, how the basic file operations are used in conjunction with other types of operations to implement typical requirements. Again, I am describing COBOL constructs and statements, but the implementation would be very similar in other high-level languages.

A common requirement involves the *display* of data from a particular record: the user identifies the record by entering the value of its key (customer number, part number, invoice number, and the like), and the application responds by retrieving that record and displaying some of its fields. When the key consists of several fields, the user must enter several values. To implement this operation in the application, all we need is a READ: we place the values entered by the user into the current key fields, perform the READ, and then display for the user various fields from the record area. If, however, the system returns an error code, we display a message such as "record not found."

If the user wants to *modify* some of the fields in a particular record, we start by performing a READ and displaying the current values, as before; but then we allow the user to enter the new values, place them in the appropriate fields in the record area, and perform a REWRITE. And if what the user wants is to *delete* a particular record, we usually start with a READ, display some of the fields to allow the user to confirm it is the right record, and then perform a DELETE.

Lastly, to *add* a record, we display blank fields and allow the user to enter their actual values. (In a new record, some fields may have null values, or some default values; so these fields may be left out, or just displayed, or displayed with the option to modify them.) The user must also enter the value of the key fields, to identify the new record. We then perform a WRITE, and the system will add this record to the file. If, however, it returns an error code, we display a message such as "duplicate key" to tell the user why the record could not be added.

[3] I will not discuss here the various *support* operations – opening and closing files, locking and unlocking records in multiuser applications, and the like. Since there is little difference between these operations in file systems and in database systems, they have no bearing on my argument. Many of these operations can be performed automatically, in fact, in both types of systems.

Examples of this type of record access are found in the *file maintenance* operations – those operations that permit the user to add, delete, and modify records in the database. And, clearly, any maintenance requirement can be implemented through the basic file operations: any file, record, and field in the database can be read, displayed, or modified. If we must restrict this freedom (permit only a range of values for a certain field, permit the addition or deletion of a record only under certain conditions, etc.), all we have to do is add appropriate checks; then, if the checks fail, we bypass the file operation and display a message.

So far I have discussed the *interactive* access of individual records, but the basic file operations are used in the same way when the user is not directly involved. Thus, if we need to know at some point in the application the quantity on hand for a certain part, we place the part number in the key field, perform a READ, and then get the value from the quantity field; if we want to add a new transaction to the sales history file, we place the appropriate values in the key fields (customer number, invoice number, etc.) and in the non-key fields (date, price, quantity, etc.), and perform a WRITE; if we want to update a customer's balance, we place the customer number in the key field, perform a READ, calculate the new value, place it in the balance field, and then perform a REWRITE. Again, any conceivable requirement can be implemented through the basic file operations.

❖

Accessing *individual* records, as described above, is one way of using indexed data files. The other way is by *scanning* records, an operation accomplished with an iterative construct based on START and READ NEXT. This construct, which may be called the basic file scanning loop, is used every time we read a series of records sequentially through an index. The best way to illustrate this loop is with a simple example (see figure 7-13). The loop here is designed to read the PART file in ascending part number sequence. The indexing key, P-KEY, consists of one field, P-NUM (part number). START positions the record pointer so that the first record read has a part number no less than P1, and the

```
  MOVE P1 TO P-NUM START PART KEY>=P-KEY INVALID GO TO L4.
L3. READ PART NEXT END GO TO L4. IF P-NUM>P2 GO TO L4.
  IF P-QTY<Q1 GO TO L3.
  [various operations]
  GO TO L3.
L4.
```

Figure 7-13

condition >P2 terminates the loop at the first record with a part number greater than P2. The loop will read, therefore, only the *range* of records, P1 through P2, inclusive.[4] In addition, within this range, the loop selects only those records where the quantity field, P-QTY, is no less than a certain value, Q1. The operations following the selection conditions will be performed for every record that satisfies these conditions. The labels L3 and L4 delimit the loop.[5]

We rarely perform the same operations with all the records in a file, so the selection of records is a common requirement in file scanning. The previous example illustrates the two selection methods – based on key fields, and on non-key fields. The method based on key fields is preferable when what we select is a range of records, as the records left out don't even have to be read. This can greatly reduce the processing time, especially if the file is large and the range selected is relatively small. In contrast, when the selection is based on non-key fields, each record in the file must be read. This is true because the value of non-key fields is unrelated to the record's position in the scanning sequence, so the only way to know what the values are is by reading the record. The two methods are often combined in the same loop, as illustrated in the example.

It should be obvious that these two selection methods are completely general, and can satisfy any requirement. For example, if the range must include all the records in the file, we specify null values for the key fields in START and omit the test for the end of the range. The loop also deals correctly with the case where no records should be selected (because there are none in the specified range, or because the selection based on non-key fields excludes all those in the range). It must be noted that the selection conditions can be as complex as we need: they can involve several fields, or fields from other files (by reading in the loop records from those files), or a combination of fields, memory variables, and constants. A complex condition can be formulated either as one complex IF statement or as several consecutive IF statements. And,

[4] Note the END clause in READ NEXT, specifying the action to take if the end of the file is reached before P2. (INVALID and END are the abbreviated forms of the COBOL keywords INVALID KEY and AT END. Similarly, GOTO can be abbreviated in COBOL as GO.)

[5] It is evident from this example that the most effective way to implement the basic file scanning loop in COBOL is with GOTO jumps. This demonstrates again the absurdity of the claim that GOTO is harmful and must be avoided (the delusion we discussed under structured programming). Modifying this loop to avoid the GOTOs renders the simple operations of file scanning and record selection complicated and abstruse; yet this is exactly what the experts have been advocating since 1970. It is quite likely that the complexity engendered by the delusions of structured programming contributed to the difficulty programmers had in using file operations, and was a factor in the evolution of database systems: because they tried to avoid the complications created by one pseudoscience, programmers must now deal with the greater complications created by another.

in addition to the conditions that affect all the operations in the loop, we can have conditions *within* the loop; any portion of the loop, therefore, can be restricted to certain records.

Let us see now how the basic file scanning loop is used to implement various file operations. In a typical file listing, or query, or report, the scanning sequence and the record selection criteria specified by the user become the index and the selection conditions for the scanning loop. And within the loop, for each record selected, we show certain fields and perhaps accumulate their values. Typically, one line is printed or displayed for each record, and the totals are shown at the end. When the indexing key consists of several fields, their value will change hierarchically, one within another, in the sorting sequence of the index; thus, we can have various levels of subtotals by noting within the loop when the value of these fields changes. In an orders file, for instance, if the key consists of order number within customer number, and if we need the quantity and amount subtotals for the orders belonging to each customer, we must show and then clear these subtotals every time the customer number changes.

Another use of the scanning loop is for *modifying* records. The reading and selection are performed as before, but here we modify the value stored in certain fields; then we perform a REWRITE (at the end of the loop, typically). This is useful when we have to modify a series of records according to some common logic. Not all the selected records need to be modified, of course; we can perform some calculations and display the results for all the records in a given range, for instance, but modify only those where the fields satisfy a certain condition. Rather than modify records, we can use the scanning loop to *delete* certain records; in this case we perform a DELETE at the end of the loop.

An interesting use of indexed data files is for sorting. If, for instance, we need a listing of certain values in a particular scanning sequence (values derived from files or from calculations), we create a temporary data file where the indexing key is the combination of fields for that scanning sequence, while the non-key fields are the other values to be listed. All we have to do then is perform a WRITE to add a record to the temporary file for each entry required in the listing. The system will build for us the appropriate index, and, once complete, we can scan the temporary file in the usual manner. Similarly, if we need to scan a portion of a data file in a certain sequence, but only occasionally, then instead of having a permanent index for that sequence we create a temporary data file that is a subset of the main data file: we read the main data file in a loop through one of its indexes, and for each selected record we copy the required fields to the record of the temporary file and perform a WRITE.

If we want to analyze certain fields in a data file according to the value present in some other fields (total the quantity by territory, total various

amounts by the combination of territory and category, etc.), we must create a temporary data file where the indexing key is the field or combination of fields by which we want to group the records (the *analysis* fields in the main data file), while the non-key fields are the values to be totaled (the *analyzed* fields). We read the main file in a loop, and, for each record, we copy the analysis values and the analyzed values to the respective fields in record of the temporary file. We then perform a WRITE for this file and check the return code. If the system indicates that the record already exists, it means this is not the first time that combination of key values was encountered; the response then is to perform a READ, *add* the analyzed values to the respective fields, and perform a REWRITE. In other words, we create a new record in the temporary file only the first time a particular combination of analysis values is encountered, and *update* that record on subsequent occasions. At the end, the temporary file will contain one record for each unique combination of analysis values. This concept is illustrated in figure 7-14.

```
    MOVE C1 TO C-NUM START CUSTOMER KEY>=C-KEY INVALID GO TO L4.
L3. READ CUSTOMER NEXT END GO TO L4. IF C-NUM>C2 GO TO L4.
    MOVE C-TER TO SR-TER MOVE C-QTY TO SR-QTY.
    WRITE SR-RECORD INVALID READ SORTFL
      ADD C-QTY TO SR-QTY REWRITE SR-RECORD.
    GO TO L3.
L4.
```

Figure 7-14

In this example, a certain quantity in the CUSTOMER file is analyzed by territory for the customers in the range C1 through C2. SORTFL is the temporary file, and SR-RECORD is its record area. The simplicity of this operation is due to the fact that much of the logic is implicit in the READ, WRITE, and REWRITE.

4

One of the most important uses of the file scanning loop is to *relate* files. If we nest the scanning loop of one file within that of another, a logical relationship is created between the two files. From a programming standpoint, the nesting of file scanning loops is no different from the nesting of any iterative constructs: the whole series of iterations through the inner loop is repeated for every iteration through the outer loop. In the inner loop we can use fields from both files; any operation, therefore, including the record selection conditions, can depend on the record currently read in the outer loop.

Figure 7-15 illustrates this concept. The outer loop scans the CUSTOMER file and selects the range of customer numbers C1 through C2. The indexing key, C-KEY, consists of one field, C-NUM (customer number). Within this loop, in addition to any other operations performed for each customer record, we include a loop that scans the ORDERS file. The indexing key here, O-KEY, consists of two fields, O-CUS (customer number) and O-ORD (order number), in this sorting sequence. Thus, to restrict the inner loop to the orders belonging to one customer, we select only the range of records where the customer number equals the one currently read in the outer loop, while allowing the order number to be any value. (Note that the terminating condition, "IF O-CUS NOT=C-NUM," could be replaced with "IF O-CUS>C-NUM," since the first O-CUS read that is not equal to C-NUM is necessarily greater than it.) The inner loop here selects *all* the orders for the customer read in the outer loop; but we could have additional selection conditions, based on non-key fields, as in figure 7-13 (for example, to select only orders in a certain date range, or over a certain amount).

```
  MOVE C1 TO C-NUM START CUSTOMER KEY>=C-KEY INVALID GO TO L4.
L3. READ CUSTOMER NEXT END GO TO L4. IF C-NUM>C2 GO TO L4.
  [various operations]
  MOVE C-NUM TO O-CUS MOVE 0 TO O-ORD.
  START ORDERS KEY>O-KEY INVALID GO TO L34.
L33. READ ORDERS NEXT END GO TO L34. IF O-CUS NOT=C-NUM GO TO L34.
  [various operations]
  GO TO L33.
L34.
  [various operations]
  GO TO L3.
L4.
```

Figure 7-15

Although most file relations involve only two files, the idea of loop nesting can be used to relate hierarchically any number of files, simply by increasing the number of nesting levels. Thus, by nesting a third loop within the second one and using the same logic, the third file will be related to the second in the same way that the second is related to the first. With two files, we saw, the second file's key consists of two fields, and the range selected includes the records where the first field equals the first file's key. With three files, the third file's key must have three fields, and the range will include the records where the first two fields equal the second file's key. (The keys may have additional fields; two and three are the minimum needed to implement this logic.)

To illustrate this concept, figure 7-16 adds to the previous example a loop to scan the LINES file (the individual item lines associated with each order).

If ORDERS has fields like customer number, order number, date, and total amount, which apply to the whole order, LINES has fields like item number, quantity, and price, which are different for each line. Its indexing key consists of customer number, order number, and line number, in this sorting sequence. And the third loop isolates the lines belonging to a particular order by selecting the range of records where the customer and order numbers equal those of the order currently read in the second loop, while the line number is any value. Another example of a third nesting level is a transaction file, where each record is an invoice, payment, or adjustment pertaining to an order, and the indexing key consists of customer number, order number, and transaction number.[6]

```
  MOVE C1 TO C-NUM START CUSTOMER KEY>=C-KEY INVALID GO TO L4.
L3. READ CUSTOMER NEXT END GO TO L4. IF C-NUM>C2 GO TO L4.
  [various operations]
  MOVE C-NUM TO O-CUS MOVE 0 TO O-ORD.
  START ORDERS KEY>O-KEY INVALID GO TO L34.
L33. READ ORDERS NEXT END GO TO L34. IF O-CUS NOT=C-NUM GO TO L34.
  [various operations]
  MOVE O-CUS TO L-CUS MOVE O-ORD TO L-ORD MOVE 0 TO L-LINE.
  START LINES KEY>L-KEY INVALID GO TO L334.
L333. READ LINES NEXT END GO TO L334.
  IF NOT(L-CUS=O-CUS AND L-ORD=O-ORD) GO TO L334.
  [various operations]
  GO TO L333.
L334.
  [various operations]
  GO TO L33.
L34.
  [various operations]
  GO TO L3.
L4.
```

Figure 7-16

Note that in the sections marked "various operations" we can access fields from all the currently read records: in the outer loop, fields from the current CUSTOMER record; in the second loop, fields from the current CUSTOMER and ORDERS records; and in the inner loop, fields from the current CUSTOMER, ORDERS, and LINES records.

Note also that the sections marked "various operations" may contain additional file scanning loops; in other words, we can have more than one

[6] Note, in figures 7-13 to 7-16, the numbering system used for labels in order to make the jumps self-explanatory (as discussed under the GO TO delusion, pp. 621–624).

scanning loop at a given nesting level. For instance, by creating two consecutive third-level loops, we can scan first the lines and then the transactions of the order read in the second-level loop.

The arrangement where the key used in the outer loop is part of the key used in the inner loop, as in these examples, is the most common and the most effective way to relate files, because it permits us to select records through their key fields (and to read therefore only a *range* of records). We can also relate files, though, by using non-key fields to select records (when it is practical to read the entire file in the inner loop).

Lastly, another way to relate files is by reading within the loop of one file just one record of another file, with no inner loop at all (or, as a special case, reading just one record in both files, with no outer loop either). Imagine that we are scanning an invoice file where the key is the invoice number and one of the key or non-key fields is the customer number, and that we need some data from the customer record – the name and address fields, for instance. (This kind of data is normally stored only in the customer record because, even though required in many operations, it is the same for all the transactions pertaining to a particular customer.) So, to get this data, we place the customer number from the currently read invoice record into the customer key field, and perform a READ. All the customer fields are then available within the loop, along with the current invoice fields.

The relationship just described, where several records from one file point to the same record in another file, is called *many-to-one* relationship. And the relationship we discussed previously, where one record from the first file points to several records in the second file (because several records are read in the inner loop for each record read in the outer loop) is called *one-to-many* relationship. These two types of file relationships are the most common, but the other two, *one-to-one* and *many-to-many*, are also important.

We have a one-to-one relationship when the same field is used as a key in two files. For example, if in addition to the customer file we create a second file where the indexing key is the customer number (in order to store some of the customer data separately), then each record in one file corresponds to one record in the other. And we have a many-to-many relationship when one record in the first file points to several records in the second one, and at the same time one record in the second file points to several records in the first one. (We will study the four types of file relationships in greater detail later; see pp. 752–755.)

To understand the many-to-many relationship, imagine a factory where a

number of different products are being built by assembling various parts from a common inventory. Thus, each product is made from a number of different parts, and at the same time a part may be used in different products. The product file has one record for each product, and the key is the product number. And the part file has one record for each part, and the key is the part number. We can use these files separately in the usual manner, but to implement the many-to-many relationship between products and parts we need an additional file – a service file for storing the cross-references. This file is a dummy data file that consists of key fields only. It has two indexes: in the first one the key is the product number and the part number, and in the second one it is the part number and the product number, in these sorting sequences. In the service file, therefore, there will be one record for each pair of product and part that are related in the manufacturing process (far more records, probably, than there are either products or parts). Now we can scan the product file in the outer loop, and the service file, through its first index, in the inner loop; or, we can scan the part file in the outer loop, and the service file, through its second index, in the inner loop. Then, by selecting in the inner loop a range of records in the usual manner, we will read in the first case the parts used by a particular product, and in the second case the products that use a particular part. What is left is to perform a READ in the inner loop using the part or product number, respectively, in order to read the actual records.

The Lost Integration

The preceding discussion was not meant to be an exhaustive study of indexed data files. My main intent was to show that any conceivable database requirement can be implemented with file operations, and that this is a fairly easy programming challenge: every one of the examples we examined takes just a few statements in COBOL. We only need to understand the two ways of using indexes (reading individual records or scanning a range of records) and the two ways of selecting records (through key fields or non-key fields). Then, simply by combining the basic file operations with the other operations available in a programming language, we can access and relate the files in the database in any way we like.

So the difficulties encountered by programmers are not caused by the basic file operations, nor by the selection of records, nor by the file scanning loops. The difficulties emerge, rather, when we combine file operations, and when we combine them with the other types of operations required by the application. The difficulties, in other words, are due to the need to deal with

interacting software structures. Two kinds of structures, and hence two kinds of interactions, are generated: one through the file relationships we discussed earlier (one-to-many, many-to-many, etc.), the other through the links created between the application's elements by the file operations.

Regarding the first kind of structures, the file relationships are easy to understand individually, because we can view them as simple hierarchical structures. If we depict the nesting of files as a structure, each file can be seen as a different level of the structure, and its records as the various elements which make up that level. The relationship between files is then the relationship between the elements of one level and the next. But, even though each relationship is hierarchical, most files take part in several relationships, through different fields. In other words, a record in a certain file can be an element in several structures at the same time, so these structures interact. The totality of file relationships in the database is a complex structure.

As for the second kind of structures, we already know that the file operations give rise to processes based on shared data (see pp. 351–353). So they link the application's elements through many structures – one structure for each field, record, or file that is accessed by several elements. Thus, in addition to the interactions due to the file relationships, we must cope with the interactions between the structures generated by file operations. And we must also cope with the interactions between these structures and the structures formed by the other types of processes – practices, subroutines, memory variables, etc. To implement database requirements we must deal with complex software structures.

When replacing the basic file operations with higher-level operations, what are the database experts trying to accomplish? All that a database system can do is replace with a built-in process the two or three statements that constitute the use of a basic file operation. The experts misinterpret the difficulty that programmers have in implementing file operations as the problem of dealing with the relatively low levels. But, as we saw, the difficulty is not due to the individual file operations, nor to the individual relationships. The difficulty emerges when we deal with *interacting* operations and relationships, and with their interaction with the rest of the application. And these interactions cannot be eliminated; we must have them in a database system too, if the application is to do what we want it to do. Even with a database system, then, the difficult part of database programming remains. The database systems can perhaps replace the easy challenges – the individual operations; but they cannot eliminate the difficult part – the need to deal with interacting structures.

What is worse, database systems make the interactions even more complex, because some of the operations are now in the application while others are in the database system. The original idea was to have database functions akin to

the functions provided by a mathematical library; that is, entities of a high level of abstraction, which interact with the application only through their input and output. But this is impossible, because database operations must interact with the rest of the application at a lower level – at the level of fields, variables, and conditions. Thus, the level of abstraction that a database system can provide while remaining a practical system is not as high as the one provided by a mathematical library. We cannot extract, for example, a complete file scanning loop, with all the operations in the loop, and move it into a database system – not if we want to retain the freedom of implementing *any* scanning loops and operations.

❖

All we needed before was the six basic file operations. The database operations, and their interaction with the rest of the application, could then be implemented with the same programming languages, and with the same methods and principles, that we use for the other operations in the application. With a database system, on the other hand, we need new and complicated principles, languages, rules, and methods; we must deal with a new kind of operations in the database system, plus a new kind of operations in the application, the latter necessary in order to link the application to the database system. So, in the end, the difficulties faced by programmers in implementing database operations are even greater than before.

It is easy to see why the basic file operations are both necessary and sufficient for implementing database operations: for most applications – business applications, in particular – they are just the right level of abstraction. The demands imposed by our applications rarely permit us to move to higher levels, and we rarely need lower ones. An example of lower-level file operations is the requirement for a kind of fields, indexes, or records that is different from the one provided by the standard data files. And, in the rare situations where such a requirement is important, we can implement it in a language like C. Similarly, in those situations where we can indeed benefit from higher-level operations, we can create them by means of subroutines in the same language as the application itself: we design the appropriate combination of basic file operations and flow-control constructs, store it as a separate module, and invoke it whenever we need that particular combination.

For the vast majority of applications, however, we need neither lower nor higher levels, since the level provided by the basic file operations is just right. This level is similar to the level provided, for general programming requirements, by our high-level languages. With the features found in a language like COBOL, for instance, we can implement any business application. Thus, it

is no coincidence that, in conjunction with the operations provided by a programming language, the basic file operations can be used quite naturally to implement practically all database operations, and also to link these operations to the other types of operations: iterative constructs are just right for scanning a data file sequentially through one of its indexes; nested iterations are just right for relating files hierarchically; conditional constructs are just right for selecting records; and assignment constructs are just right for moving data between fields, and between fields and memory variables. It is difficult to find a single database operation that cannot be easily and naturally implemented with the constructs found in the traditional languages.

This flexibility is due to the correct level of abstraction of both the basic file operations and the traditional languages. This level is sufficiently low to make all conceivable database operations possible, and at the same time sufficiently high to make them simple and convenient – for an experienced programmer, at least. We can so easily implement any database requirement using ordinary features, available in most languages, that it is silly to search for higher-level operations.

High-level database operations offer no benefits, therefore, for two reasons: first, because we can so easily implement database requirements using the basic file operations, and second, because it is impossible to have built-in operations for all conceivable situations. No matter how many high-level operations we are offered, and no matter how useful they are, we will always encounter requirements that cannot be implemented with high-level operations alone. We cannot give up the lower levels, thus, because we need them to implement details, and because the links between database operations, and also between database operations and the other types of operations, occur at the low level of these details.

So the idea of higher levels is fallacious for database operations in the same way it is fallacious for the other types of operations. This was also the idea behind the so-called fourth-generation languages (see pp. 464–465). And, like the 4GL systems, the relational systems became in the end a fraud.

The theorists start by promising us higher levels. Then, when it becomes clear that the restriction to high levels is impractical, they restore – in the guise of enhancements – the low levels. Thus, with 4GL systems we still use such concepts as conditions, iterations, and assigning values to variables; in other words, concepts of the same level of abstraction as those found in a traditional language. It is true that these systems provide *some* higher-level operations (in user interface, for instance), but they do not eliminate the lower levels. In any case, even in those situations where operations of a higher level are indeed useful, we don't need these systems; for, we can always provide the higher levels ourselves, in any language, through subroutines. Similarly, we will see in the

present section, the relational database systems became practical only after restoring the low levels; that is, the traditional file management concepts.

In conclusion, the software elites promote ideas like 4GL and relational databases, not on the basis of any real benefits, but in order to deprive us of the programming freedom conferred by the traditional languages. Their real motive is to force us to depend on expensive and complicated development systems, which they control.

I want to stress again that remarkable quality found in the basic file operations, the fact that they are at the same level of abstraction as the operations provided by the traditional programming languages. This is why we can so easily link these operations and implement database requirements. One of the most successful of all software concepts, this simple feature greatly simplifies both programming and the resulting applications.

There is a *seamless integration* of the database and the rest of the application, for both data and operations. The fields, the record area, and the record keys function as both database entities and memory variables at the same time. Database fields can be mixed freely with memory variables in assignments, calculations, or comparisons. Transferring data between disk and memory is a logical extension of the data transfers performed in memory. Most statements, constructs, and methods we use in programming have the same form and meaning for file operations as they have for the other types of operations; iterative and conditional constructs, for example, are used in the same way to scan and select records from a file as they are to scan and select items from an array or table stored in memory.

Just by learning to use the six basic file operations, then, a programmer gains the means to design and control databases of any size and complexity. The most difficult part of this work is handled by the file management system, and what is left to the programmer is not very different from the challenges he faces when dealing with any other aspect of the application.

The seamless integration of the database and the application is such an important feature that, had we not already had it in the traditional file operations, we could have rightly called its introduction today a breakthrough in programming techniques. The ignorance of the academics and the practitioners is betrayed, thus, by their lack of appreciation of a feature that has been widely available (through COBOL, for instance) since the 1960s. Instead of studying it and learning how to make the most of it, the software experts have been promoting the relational model, whose express purpose is to *eliminate* the integration. In their attempt to simplify programming, they restrict the links

between files, and between files and the rest of the application, to high levels of abstraction. But this is an absurd idea, as we saw, because serious applications require low-level links too.

Then, instead of admitting that the relational model had failed, the experts proceeded to *reestablish* the low-level links. For, in order to make the relational model practical, they had to restore the integration – the very quality that the relational model had tried to eliminate. But the only way to provide the low levels and the integration now, as part of a database system, is through a series of artificial enhancements. When examined, the new features turn out to be nothing but particular instances of the important quality of integration: they are means to link the database to the rest of the application in specific situations. What is the very nature of the traditional file operations, and in effect just one simple feature, is now being restored by annulling the relational principles and replacing them with a multitude of complicated features. Each new feature is, in reality, a substitute for a particular high-level software element (a particular database function) that can no longer be implemented naturally, by combining lower-level elements.

Like all development systems that promise a higher level of abstraction, the relational systems became increasingly large and complicated because they attempted to replace with built-in operations the infinity of alternatives that we need at high levels but can no longer create by starting from low levels. Recall the analogy of software with language: If we had to express ourselves through ready-made sentences, instead of creating our own starting with words, we would end up depending on systems that become increasingly large and complicated as they attempt to provide all necessary sentences. But even with thousands of sentences, we would be unable to express all possible ideas. So we would spend more and more time trying to communicate through these systems, even while being restricted to a fraction of the ideas that can be expressed by combining words.

Thus, the endless problems engendered by relational database systems, and the astronomic cost of using them, are due to the ongoing effort to overcome the restrictions imposed by the relational model. They are due, in the end, to the software experts, who not only failed to understand why this model is worthless, but continued to promote it while its claims were being falsified.

The relational model became a pseudoscience when the experts decided to "enhance" it, which they did by turning its falsifications into features (see p. 225); specifically, by restoring the traditional data management concepts. It is impossible, however, to restore the seamless integration we had before. So all we have in the end is some complicated and inefficient database systems that are struggling to emulate the simple, straightforward file systems.